Minimal Spanning Trees, Greedy Algorithms, and Exchange Arguments

Suhas Arehalli

COMP221 - Spring 2024

1 Definitions

- Suppose we have a *weighted* undirected graph, which we will formally represent as an unweighted graph $G = (V, E)$ of vertices and edges with a *weight* or *cost function* $c : E \to \mathbb{R}$ that maps each edge $e \in E$ to a real-valued weight $c(e) \in \mathbb{R}$.
  - Weighted graphs can be formalized in different ways too! For instance, we can have edges be 3-tuples: $G = (V, E)$, with V the set of vertices and E the set of edges with $(v_1, v_2, c) \in E$ representing an edge from $v_1 \in V$ to $v_2 \in V$ with weight c.
  - We could also choose to call an unweighted graph $G = (V, E)$ and a weighted graph $G = (V, E, c)$ with $c : E \to \mathbb{R}$ still a cost function.
  - All of these formalizations are equivalent, so I will choose the one above because it's the one that's most intuitive to me (and will make the transition to Flow Networks easiier. You can swap out the details below for the one that makes the most sense to you, but you will probably encounter a variety of formalizations in different courses so getting used to the variation is helpful!
- What we're interested in are **Minimal Spanning Tree**s, or MSTs. An MST is...
  1. A *Sub-graph* of G, $M = (V', E')$, with $V' \subseteq V$ and $E' \subseteq E$.
  2. *Spanning*: $V' = V$ (i.e., the subgraph contains all of the vertices in G)
  3. A *Tree*: M is connected (there is a path between any two vertices in M) and acyclic (M contains no cycles).
  4. *Minimal*: We want to minimize the total cost of M, which we define as the sum of the weights of the edges we include: $\sum_{e \in E'} c(e)$. That is, for any **Spanning Tree** that fits the above criteria, we want the one that has the minimal cost.

2 Prim's Algorithm

2.1 The Core Idea & Greedy Algorithms

- Note that since we must have a *spanning* tree at the end of the process, the set of vertices in M must be exactly V!

- Then we can reframe the problem of constructing an MST to the problem of selecting edges from E!
- In Prim's algorithm we will do this sequentially, choosing edges one by one.
  - At the end of the process, we will need to have a minimal spanning *tree*, and so in Prim's our idea is to choose edges that maintain the tree structure (i.e., acyclic and connected) of M as we build it.
  - Note that adding an edge with both vertices within M already would create a cycle, and that adding an edge with both vertices outside of M would make our graph disconnected!
  - Thus we add only edges where one vertex is already in V' and one vertex is in $V \setminus V'$.
  - We stop when we're spanning, so we we can guarantee that we'll have a spanning tree!
  - We also want the spanning tree we find to have minimal cost. The idea of Prim's is to do this in a **greedy** fashion: We will choose the *lowest cost* edge that satisfies the properties above!
- We apply these principles and get this pseudocode for Prim's algorithm (Alg. **??**

Algorithm 1 Pseudocode for Prim's Algorithm

function PRIMS($G = (V, E)$, c, s)
  $V' \leftarrow \{s\}$
  $E' \leftarrow \emptyset$
  while $|V'| < |V|$ **do**
    Let $X = \{(u, v) \in E \mid u \in V', v \in V \setminus V'\}$
    $e = (u, v) \leftarrow \arg\min_{(u,v)\in X} c(u, v)$
    $E' \leftarrow E \cup \{e\}$
    $V \leftarrow V \cup \{v\}$
  end while
  return $M = (V', E')$
end function

- Note that for convenience, we assume that G is connected. Otherwise, no MST exists, and you likely want to find a connected component to run Prim's on.
- Prim's falls into the class of **Greedy Algorithms**.
  - Algorithmic problems often take the form of sequential decision problems
  - A greedy solution will always make the locally optimal choice
  - For many problems, a greedy solution is not optimal! An early optimal choice locks us out of the globally optimal solution later on!
  - However, when greedy solutions *are* optimal (i.e., where the locally optimal choices lead to globally optimal solutions), greedy solutions are typically very efficient!
  - This means that proofs of correctness for greedy algorithms can be particularly tricky.
  - We have a strategy for doing this called an **Exchange Argument**.

* Our goal will be to prove by *induction* that, at each decision, the locally optimal choice does not lock us out of the globally optimal solution.
* We prove the inductive step by *contradiction* — We assume we are locked out of any globally optimal choice and consider an optimal choice that was available to us before we made this decision (which exists via out inductive hypothesis).
* We reach a contradiction by constructing an *exchange*. We construct a new solution by exchanging a piece of the presumed optimal solution with the piece that came from the presumed incorrect decision we made. We then show that this constructed solution is as optimal or better than the presumed optimal solution, which contradicts our assumption that our decision locked us out of an optimal solution later on — we just showed a solution that must be optimal that incorporates our decision!
* We then conclude that since the inductive argument has shown none of our decisions can lead us away from the optimal solution, once we arrive at a solution, it must be the optimal one.

2.2 Proof of Correctness (Prim's

Let's proceed by induction:

Statement. E' is always a subtree of an MST of G.

To see that we always build a tree, we can simply observe that the only edges we consider adding have one edge in M and one edge outside of M. This cannot create a cycle, since before adding the edge one vertex is not connected to any vertex in M, and M is connected afterward since after being added one of the two vertices was already within M, and thus connected to every over vertex in M. Thus the new vertex in the new edge must be connected since there is a path through the the vertex that was in M previously to every other vertex in M. It might be helpful to write out this part of the inductive proof more carefully as an exercise, but it won't be the focus of these notes!

Now let's focus on the tricky part: We never pick an edge not in the MST! Proceed by induction over insertions into E'

Base Case: E is empty, and trivially a subtree of any MST's edge set.

Inductive Step: Now, we get to the exchange bit. Proceed by contradiction.

By our inductive hypothesis, we know that before this insertion, $\exists M^* = (V, E^*)$, an MST of G such that $E' \subseteq E^*$.

We assume for contradiction that (u, v) is not an edge in *any* of these MSTs. WLOG lets refer to one of these MSTs $M^* = (V, E^*)$ s.t. $E' \subseteq E^*$.

Since M^* spans G, $u, v \in V^*$, and since M^* is a tree, it must be connected, and thus there exists a path P through M^* that connects u to v. Since $(u, v) \notin E^*$, this path crucially does not contain (u, v)! Since $u \in V'$ and $v \in V \setminus V'$, there must be an edge $e' = (u', v')$ along P such that $u' \in V$ and $v' \in V$.[1]

Now we get to our exchange: Construct $E'^* = E^* \cup \{(u, v)\} \setminus \{(u', v')\}$. I will claim now that $M'^* = (V, V'^*$ is a spanning tree. Spanning because we include all the vertices, which should be

[1] You can prove this formally by contradiction if you need to: Suppose there was no such edge. Since we know every edge is in V and the path begins at $u \in V'$, we can prove by induction over the number of edges in the path that $v \in V'$, which is a contradiction. Informally, if we never take an edge that leads outside of V', our path must end in V'. Thus if we end outside of V' (i.e., in $V \setminus V'$), we must have taken an edge out of V'

straightforward. Acyclic and connected is a bit trickier, but involves understanding that if M^* was a spanning tree, adding (u, v) creates a cycle and removing (u', v') removes the cycle while preserving connectivity. This is simple enough to prove, so i'll leave it as an exercise[2] To visualize what's going on see Fig. **??** for a visual of a particular case of this scenario.

Now, let's look at the cost of $M'*$! I'll move to using e and e' to label our edges for convenience, since the vertex names are no longer relevant. By definition, the cost of M'^* is

$$\begin{aligned} c(M'^*) &= \sum_{f \in M^* \cup \{e\} \setminus \{e'\}} c(f) \\ &= \sum_{f \in M^*} +c(e) - c(e') \\ &= c(M^*) + c(e) - c(e') \end{aligned}$$

Now let's consider one last fact: Prim's chose e because it had the minimum cost in our viable edge set X. Then observe that we constructed e' such that $e' \in X$ as well! Thus, we know $c(e) \leq c(e')$, and thus $c(M'^*) \geq c(M)$ — our constructed MST M'^* is equal to or better than our presumed optimal MST M^*. Since we can't have a spanning tree with cost less than an MST, we are left to assume that M'^* is an MST as well. However, this still contradicts our assumption that $e = (u, v)$ is not an edge in *any* MST with edges E', since M'^* has $E' \cup \{e\} \subseteq E'^*$! Thus, E' is always a subset of the edge set of an MST!

After the last iteration of the loop, we have that $V' = V$, so $M' = (V', E')$ is a spanning tree, and so if $E' \subseteq E^*$ for some MST's edge set E^*, then $E = E^*$ and we have an MST.

2.3 Runtime Analysis

We won't talk too much about efficiency, except to highlight the role of DS design in making Prim's fast. Naively, finding an edge that satisfies the properties we require would involve iterating through all edges at each pass, leading to a quadratic algorithm over the graph: something $O(|V||E|)$, assuming $E \geq V - 1$ since we assume the graph is connected. A more clever solution involves managing a priority queue that allows for updating a value's priority within the queue efficiently. With careful implementation, we can have a PQueue that has, at maximum, one entry per unreached vertex, leading to a $O(|E| + |V| \log |V|)$ time complexity — much better for large graphs!

3 Kruskal's Algorithm

3.1 The Core Idea

Like Prim's, our goal is to build our MST in a greedy way, edge by edge. The key difference is that we will weaken our requirements for our partial solutions: Rather than building up a tree — a connected, acyclic graph — we'll be building a **forest** — a graph that only needs to be acyclic!

This puts a slightly different restriction on the edges under consideration. They no longer need to preserve connectedness, so we need only ensure that we don't select an edge that connects two

[2]For connectivity: How might you build a path from one vertex to any other vertex if you previously used (u', v')? take a detour to (u, v)! For acyclicity, suppose adding (u, v) added a cycle that doesn't include (u', v'). Prove that that cycle must have also existed in M^*, a contradiction.

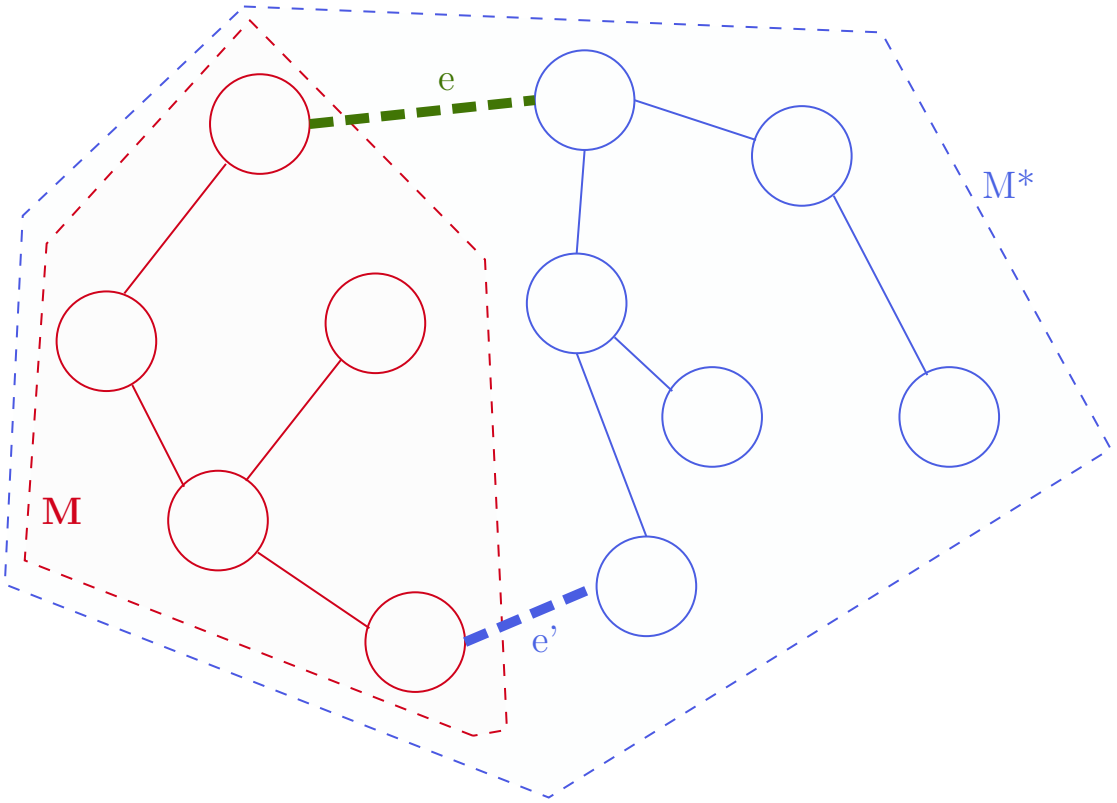

Figure 1: An illustration of the exchange argument at the core of the proof of correctness for Prim's Algorithm. We are constructing the tree in red, M, which so far is a subgraph of the MST M^*, and have chosen to select the edge e to add. we suppose for contradiction that the MST actually selects edge e' that connects M to the rest of M^*.

things that are already connected, since that will create a cycle. Otherwise, we just adopt a greedy strategy: pick the lowest cost edge that connects two disconnected vertices!

Thus we get the following pseudocode:

Algorithm 2 Kruskal's algorithm.

```
function KRUSKAL(G = (V, E))
    E' ← ∅
    while |E'| < |V| − 1 do
        Let X = {(u, v) ∈ E | u, v disconnected in(V, E')}
        e = (u, v) ← arg min_{(u,v)∈X} c(u, v)
        E' ← E' ∪ {e}
    end while
    return M = (V, E')
end function
```

4 Optimization and Runtime Analysis

As presented, both versions of Kruskal's can't be analyzed yet: We need to have a way to determine whether two vertices are connected in our forest!

One way is a graph traversal — BFS or DFS — that would actually search for the path between two vertices. This would take graph-linear time ($O(|V| + |E|)$) for each pair, which will quickly get expensive. If we're clever, we can sort our edge list in advance, ensuring that we, at worst, make $|E|$ iterations in our loop, each of which would require a $O(|V| + |E|)$ DFS. Assuming a sufficiently dense graph to be connected, this gets us to a $O(|E|^2)$ algorithm. We can do better!

4.1 Union-Find

Consider a hypothetical data structure that could help us more efficiently test for connectedness. Since we don't actually need the full path from one vertex to another, it seems reasonable (and in fact is reasonable!) to hope there's a faster way to get just connectedness information.

A HashMap that maps vertices to a representative vertex label would be very good at FIND — $O(1)$ amortized look-ups! But it would be fairly bad at union — $\Theta(n)$ to go through the entire HashTable to update each vertex's representative vertex!

A better idea is to build a parallel graph structure that manipulates edges to preserve connectedness, but make look-ups faster. That's the Union-Find algorithm! Here's the ADT we want:

FIND(v) Map a vertex to a value representing the connected component it belongs to. We will have this be a representative vertex in that connected component.

UNION(v_1, v_2) An operation that updates the data structure to reflect that an edge was added that merges two connected components.

To implement this, we build a forest! Each tree in our forest represents a connected component, represented by the root vertex of the tree.

To FIND the representative vertex, we just need to follow parent links to get to the root. This should be nice and fast if our trees are balanced — $O(\log n)$.

To UNION, we need to merge the trees that contain each vertex in the argument. Since these trees need not be binary, we can simply make one tree's root the child of the root of the other tree. Which one should be the child? The one with lesser height, since this minimizes the change in the total height of the tree! Here's a short argument:

Let h_1 be the height of taller tree T_1 with n_1 total nodes, and h_2 be the height of the shorter tree T_2 with n_2 total nodes. If $h_2 < h_1$, then making T_2 a child of T_1 will not increase the height, because the depth of every vertex in T_2 will increase by exactly 1, which cannot lead to a height greater than $h_2 + 1 \leq h_1$. If $h_1 = h_2$, then the height of the merged tree will be exactly $h_1 + 1$. With this, we can show that if the $h_1 \leq \log_2 n_1 + 1$ and $h_2 \leq \log_2 n_2 + 1$, then, if $h_1 > h_2$, we can see that

$$h_1 \leq \log_2 n_1 + 1 \leq \log_2(n_1 + n_2) + 1$$

indicating our tree is still balanced. If $h_1 = h_2$, then we have that

$$\begin{aligned} h_1 = h_2 &\leq \log_2 \min(n_1, n_2) + 1 \\ &\leq \log_2 2min(n_1, n_2) \leq \log_2(n_1 + n_2) \\ h_1 + 1 &\leq \log(n_1 + n_2) + 1 \end{aligned}$$

again indicating that we're still balanced!

This gives us a fast way to do a FIND — follow parent points up to the root in $O(\log n)$ time — and UNION — call FIND on both vertices and set the smaller tree to be the child of the parent. Since each operation can be done in $O(\log n)$ time, we can check for connectedness (i.e., does FIND$(u) =$ FIND(v)) and update our Union-Find data structure in $O(\log n)$ time in each iteration of Kruskal's!

A particularly efficient implementation can do this with $|V|$ space, since we need only maintain an array with parent pointers, as well as some information about tree heights and so on to make UNION efficient. See Skiena for details!